Model-View-Controller (MVC)

Table of contents

The three components 1
Flow of control 2
Why use it? 3
Minimal skeleton 3
Smells that break the pattern 4
In practice 4

Model-View-Controller (MVC) is a design pattern that separates an application into three interconnected components, each with a single, distinct responsibility.

- **Model** — stores the *data* and *business rules* of the application; responsible for **every state change**.
- **View** — some (usually visual) *representation* of the data, **without altering it**.
- **Controller** — an *interface* that receives *user input* (commands), tells the Model what to do, then selects a View to present the result.

The asymmetry in that third bullet is the pattern: the Controller is the only component that knows about both of the others. The View does not read from the Model, and the Model does not know about either of the others.

The three components

Layer	Core question	Owns	MUST NOT
Model	What *is* the data?	Data & rules	`print`, parse args, call `input()`
View	How is info *shown*?	Formatting & rendering	change state, validate data
Controller	What happens *next*?	Workflow & commands	store raw data, format output

Model knows nothing about the outside world.

The Model never prints, reads input, or otherwise interacts with the user directly — it is pure data and logic.

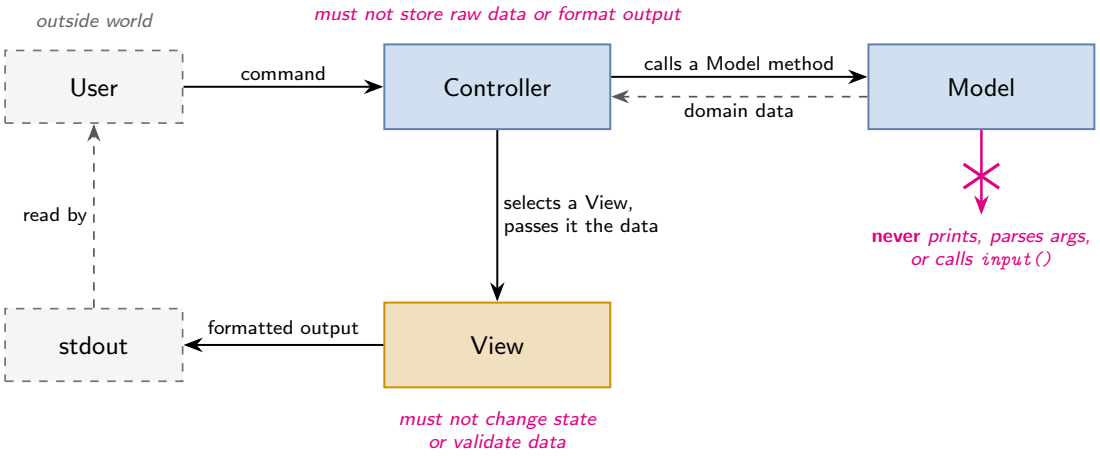

Figure 1: The dependency shape the three-component table implies. Solid arrows are calls; the two dashed arrows are not calls but returns - the Model's domain data going back to the Controller and not to the View, and the output the User reads off stdout; the dashed grey boxes are the outside world, not classes. The magenta text beside each layer is its MUST NOT rule, and the crossed magenta stub under the Model marks that it has no route to the outside world. The box fill colours carry no meaning.

Flow of control

1. **User** issues a command.
2. **Controller** parses the input and calls the appropriate **Model** method.
3. **Model** mutates its own state and returns domain data.
4. **Controller** selects a **View** and passes it the data.
5. **View** formats the data and presents it (e.g. prints to stdout).
6. Control returns to the main loop, waiting for the next user action.

Two steps are routinely misremembered. Step 3 returns the domain data **to the Controller**, not to the View — that's why the return arrow in Figure 1 points back left. And step 4 means the View is chosen **after** the Model has already run, which leaves room for which View gets used to depend on what the Model returned.

See 2025-10-14-model-view-controller[1] for the same six steps against a worked example, drawn out step by step — including the one place that example's code departs from them.

Why use it?

- **Modularity** — swap in a new model, view, or controller without breaking the rest of the application.
- **Testability** — each component can be tested in isolation.
- **Separation of concerns** — each component has exactly one job.

The Model is the easiest of the three to test, because the table above forbids it any I/O at all. Modularity in particular depends on one implementation habit: the Controller should be **given** its Model and View rather than constructing them itself, so that swapping either is a change at the call site and not a change inside the Controller. The worked example does this: its run loop builds `TodoApp(TodoList(), TodoView())` and hands both in.

Minimal skeleton

This is the worked example with the todo-list domain taken out, so that the shape of the three classes is all that is left.

```
class Model:
    def __init__(self):
        self._state = ...

    def do_something(self, ...):
        # mutate self._state, return domain data
        ...

class View:
    def show(self, data):
        # format `data` and present it -- no mutation, no logic
        ...

class Controller:
    def __init__(self, model, view):
        self._model = model
        self._view = view

    def handle(self, command):
```

[1] courses/csse1001/2025-10-14-model-view-controller.pdf

```
        # parse `command`, call the right Model method,
        # then pass its result to the View
        ...

# the run loop belongs with the Controller
controller = Controller(Model(), View())
while True:
    command = input("$ ")
    if command.strip() in ("quit", "exit"):
        break
    controller.handle(command)
```

The skeleton has no `print` in `Model`, no `input()` anywhere but the run loop, and no reference to `Controller` from inside `View` or `Model`.

Smells that break the pattern

Warning signs that a layer has taken on work belonging to another. Each row is one cell of the MUST NOT column above, turned into something you can spot while reading code.

Smell	Which rule it breaks
A `print` or an f-string inside a Model method	Model must not format output
A Model method that calls `input()` to ask a follow-up question	Model must not read input
A View method that assigns to an attribute of the object it was passed	View must not change state
A View that rejects bad data instead of displaying it	View must not validate
A Controller that keeps its own copy of the task list, scores, etc.	Controller must not store raw data
A Controller that builds the output string itself and prints it	Controller must not format output

In practice

The worked example keeps the three layers in three separate classes (a todo list manager: `Task`/`TodoList` as Model, `TodoView` as View, `TodoApp` as Controller). Real GUI and game applications often merge View and Controller into a single class, since rendering and input handling both live inside the same per-frame/per-event loop. The property worth preserving

even then is the Model's isolation: whatever owns rendering and input, the Model itself should stay free of `print`/`input()`/UI-framework calls.

Small teaching examples frequently bend the Controller and View rules too — the worked exercise in 2025-10-14-model-view-controller[2] has its Controller printing status messages directly and its View reading private attributes off Model objects — which is worth knowing about, because "does this obey the table?" is a much sharper question than "does this look like MVC?".

Numerous variants of MVC exist. Treat the version above as the base form.

See python-inheritance[3] for abstract base classes and MRO, and python-composition[4] for is-a vs has-a.

[2] courses/csse1001/2025-10-14-model-view-controller.pdf
[3] courses/csse1001/python-inheritance.pdf
[4] courses/csse1001/python-composition.pdf